7 Сети, сокеты и безопасность

Основа приложения Node неизменно зависит от двух главных компонентов инфраструктуры: сетевой поддержки и безопасности. А о сетях невозможно говорить без упоминания сокетов.

Я объединяю темы сетей и безопасности, потому что с выходом за пределы одной изолированной машины безопасность должна постоянно оставаться вашей первоочередной заботой. Каждый раз, когда вы завершаете работу над новым компонентом приложения, прежде всего следует спросить себя: насколько он безопасен? Никакое элегантное программирование не поможет, если в вашу систему проникнет вредоносный код.

Серверы, потоки и сокеты

Большая часть базовых API Node относится к созданию сервисов, прослушивающих конкретные типы коммуникаций. В главах 1 и 5 мы использовали модуль `HTTP` для создания веб-серверов, прослушивающих запросы HTTP. Другие модули могут создавать серверы TCP (Transmission Control Protocol), серверы TLS (Transport Layer Security) и сокеты UDP (User Datagram Protocol). О TLS я расскажу позднее в этой главе, а сейчас хочу познакомить вас с базовой функциональностью TCP и UDP в Node. Впрочем, сначала необходимо сказать пару слов о сокетах.

Сокеты и потоки

Сокет (socket) представляет собой конечную точку обмена данными, а *сетевой сокет* — конечную точку обмена данными между приложениями, работающими

на двух разных компьютерах в сети. Данные, передаваемые между сокетами, образуют *поток* (stream). Данные в потоке могут передаваться либо в виде двоичных данных в буфере, либо в виде строки в кодировке Юникод. Оба типа данных передаются в форме *пакетов*: частей данных, разделенных на блоки сходного размера. Также существует специальная разновидность пакетов — завершающий пакет (FIN), отправляемый сокетом как сигнал о завершении передачи.

Представьте двух людей, говорящих по портативной радиосвязи. Рации соответствуют конечным точкам передачи данных — сокетам. Когда два (или более) человека хотят поговорить друг с другом, они должны настроиться на одну частоту. Затем один участник нажимает кнопку на своей рации и соединяется с другим участником, используя некую форму идентификации. Каждый участник произносит слово «прием», сообщая, что он закончил говорить и теперь перешел к прослушиванию. Другой участник нажимает кнопку разговора на своей рации, произносит свою реплику и тоже говорит «прием», сообщая, что он переходит в режим прослушивания. Разговор продолжается до тех пор, пока одна из сторон не скажет «конец связи», сообщая о завершении сеанса. В любой момент времени говорить может только один человек.

Такие коммуникационные потоки называются *полудуплексными*, потому что передача в любой момент осуществляется только в одном направлении. *Дуплексные* потоки поддерживают двустороннюю передачу данных.

Эти концепции также относятся к потокам Node. Мы также работали с дуплексными и полудуплексными потоками в главе 6. Потоки, использованные для записи и чтения файлов, были полудуплексными: потоки поддерживали либо интерфейс чтения, либо интерфейс записи, но не оба сразу. Поток сжатия zlib является примером дуплексного потока, так как он поддерживает одновременно и чтение и запись. А теперь мы возьмем всю эту информацию и применим ее к сетевым (TCP) и зашифрованным (`Crypto`) потокам. Мы рассмотрим модуль `Crypto` позднее в этой главе, но сначала займемся TCP.

Серверы и сокеты TCP

Протокол TCP предоставляет коммуникационную платформу для большинства интернет-приложений, включая веб-службы и электронную почту. Он предоставляет механизм надежной передачи данных между клиентскими и серверными сокетами. TCP обеспечивает инфраструктуру, на которой строится прикладной уровень (например, протокол HTTP).

Клиент и сервер TCP создаются точно так же, как это делалось с HTTP, с небольшими различиями. При создании сервера TCP вместо передачи функции создания сервера `requestListener` с отдельными объектами ответа и запроса в единственном аргументе функции обратного вызова TCP передается экземпляр сокета, способного к отправке и получению данных.

Чтобы вы лучше поняли, как работает TCP, в листинге 7.1 представлен код создания сервера TCP. После создания серверный сокет прослушивает два события: получение данных и закрытие соединения клиентом. Затем полученные данные выводятся на консоль и отправляются обратно клиенту.

Сервер TCP также присоединяет обработчик для событий `listening` и `error`. В предыдущих главах я просто выводила сообщение `console.log()` после создания сервера, следуя сложившейся практике в Node. Но поскольку событие `listen()` асинхронно, с технической точки зрения эта практика неверна — сообщение выводится еще перед возникновением события. Вместо этого можно встроить сообщение в функцию обратного вызова функции `listen` или же сделать то, что я делаю здесь: связать обработчик с событием `listening` и правильно предоставить обратную связь.

Кроме того, я также реализую более сложную обработку ошибок по образцу, представленному в документации Node. Приложение обрабатывает событие `error`; если ошибка произошла из-за того, что порт в настоящее время используется, то приложение некоторое время ожидает, а потом пробует снова. Для других ошибок, например обращений к порту 80, требующему специальных привилегий, на консоль выводится полное сообщение об ошибке.

Листинг 7.1. Простой сервер TCP с сокетом, прослушивающим клиентскую передачу данных на порте 8124

```
var net = require('net');
const PORT = 8124;

var server = net.createServer(function(conn) {
   console.log('connected');

   conn.on('data', function (data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
        console.log('client closed connection');
```

```
    });

}).listen(PORT);

server.on('listening', function() {
    console.log('listening on ' + PORT);
});

server.on('error', function(err){
  if (err.code == 'EADDRINUSE') {
    console.warn('Address in use, retrying...');
    setTimeout(() => {
      server.close();
      server.listen(PORT);
    }, 1000);
  }
  else {
    console.error(err);
  }
});
```

При создании сокета TCP можно передать дополнительный объект с параметрами, состоящий из двух значений: `pauseOnConnect` и `allowHalfOpen`. По умолчанию оба свойства имеют значение `false`:

```
{ allowHalfOpen: false,
pauseOnConnect: false }
```

Присваивание `allowHalfOpen` значения `true` запрещает сокету отправлять пакет FIN при получении пакета FIN от клиента. В этом случае сокет остается открытым для записи (не для чтения), и для закрытия сокета необходимо использовать функцию `end()`. Присваивание `pauseOnConnect` значения `true` позволяет принять соединение без чтения данных. Чтобы приступить к чтению данных, следует вызвать метод `resume()` для сокета.

Для тестирования сервера используйте клиентское приложение TCP, например утилиту `netcat (nc)` в Linux или OS X или эквивалентное приложение Windows. Следующая команда подключается к серверному приложению на порте 8124 и записывает данные из текстового файла на сервер:

```
nc burningbird.net 8124 < mydata.txt
```

В Windows для тестирования можно воспользоваться такими инструментами TCP, как SocketTest (*http://sockettest.sourceforge.net/*).

Вместо того чтобы использовать программу для тестирования сервера, вы можете создать собственное клиентское приложение TCP. Как видно из листинга 7.2, клиент TCP создается так же просто, как и сервер. Данные передаются в буфере, но мы можем использовать `setEncoding()` для чтения данных в виде строки `utf8`. Метод `write()` сокета используется для передачи данных. Клиентское приложение также назначает функции прослушивания для двух событий: `data` для получения данных и `close` на случай, если сервер закроет соединение.

Листинг 7.2. Клиентский сокет отправляет данные серверу TCP

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');

// Подключение к серверу
client.connect ('8124','localhost', function () {
   console.log('connected to server');
   client.write('Who needs a browser to communicate?');
});

// Полученные данные отправляются серверу
process.stdin.on('data', function (data) {
   client.write(data);
});

// Возвращаемые данные выводятся на консоль
client.on('data',function(data) {
   console.log(data);
});

// При закрытии сервера
client.on('close',function() {
   console.log('connection is closed');
});
```

Данные, передаваемые между двумя сокетами, вводятся в терминале и передаются при нажатии клавиши Enter. Клиентское приложение отправляет введенный текст, а сервер TCP выводит его на консоль при получении. Сервер возвращает полученное сообщение клиенту, который в свою очередь выводит сообщение на свою консоль. Сервер также выводит IP-адрес и порт клиента, используя свойства `remoteAddress` и `remotePort` сокета. Ниже приведен консольный вывод сервера после получения нескольких строк от клиента:

```
Hey, hey, hey, hey-now.
 from ::ffff:127.0.0.1 57251
Don't be mean, we don't have to be mean.
 from ::ffff:127.0.0.1 57251
Cuz remember, no matter where you go,
 from ::ffff:127.0.0.1 57251
there you are.
 from ::ffff:127.0.0.1 57251
```

Соединение между клиентом и сервером поддерживается до того момента, пока вы не уничтожите одного из участников клавишами Ctrl+C. Сокет, оставшийся открытым, получает событие `close`, сообщение о котором выводится на консоль. Сервер может обслуживать несколько соединений от нескольких клиентов, поскольку все соответствующие функции асинхронны.

ПРЕОБРАЗОВАНИЕ АДРЕСОВ IPV4 В АДРЕСА IPV6

Вывод, полученный при запуске пары приложений TCP в этом разделе, показывает, что адрес IPv4 преобразуется в адрес IPv6 добавлением ::ffff.

Вместо того чтобы связывать порт с сервером TCP, мы можем связать его напрямую с сокетом. Для демонстрации этой возможности я изменила сервер TCP из предыдущих примеров, но новый сервер связывается с *сокетом Unix* (листинг 7.3). Сокет Unix соответствует некоторому пути на вашем сервере. Для более точного управления доступом к сокету могут использоваться разрешения на чтение и запись, вследствие чего этот метод является более предпочтительным, чем интернет-сокеты.

Мне также пришлось внести изменения в обработку ошибок, чтобы сокет Unix удалялся при перезапуске приложения, если сокет уже используется. В реальном приложении перед выполнением столь радикальных действий следует убедиться в том, что сокет не используется другими клиентами.

Листинг 7.3. Связывание сервера TCP с сокетом Unix

```
var net = require('net');
var fs = require('fs');

const unixsocket = '/somepath/nodesocket';

var server = net.createServer(function(conn) {
   console.log('connected');
```

```
   conn.on('data', function (data) {
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
        console.log('client closed connection');
   });

}).listen(unixsocket);

server.on('listening', function() {
    console.log('listening on ' + unixsocket);
});

// При выходе с перезапуском сервера следует удалить сокет
server.on('error',function(err) {
   if (err.code == 'EADDRINUSE') {
      fs.unlink(unixsocket, function() {
          server.listen(unixsocket);
      });
   } else {
      console.log(err);
   }
});

process.on('uncaughtException', function (err) {
    console.log(err);
});
```

Я также использую `process` для дополнительной защиты от исключений, не обработанных приложением.

ПРОВЕРКА НАЛИЧИЯ ДРУГОГО ЭКЗЕМПЛЯРА СЕРВЕРА

Прежде чем удалять сокет, необходимо проверить, не выполняется ли другой экземпляр сервера. Решение, опубликованное на сайте Stack Overflow (*http://stackoverflow.com/questions/16178239/gracefully-shutdown-unix-socket-server-on-nodejs-running-under-forever*), представляет альтернативный способ освобождения ресурсов для подобных ситуаций.

Клиентское приложение приведено в листинге 7.4. Оно принципиально не отличается от более ранней версии клиента, работавшей с портом. Изменения относятся только к точке соединения.

Листинг 7.4. Подключение к сокету Unix и вывод полученных данных

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');

// Подключение к серверу
client.connect ('/somepath/nodesocket', function () {
   console.log('connected to server');
   client.write('Who needs a browser to communicate?');
});

// Полученные данные отправляются серверу
process.stdin.on('data', function (data) {
   client.write(data);
});

// Возвращаемые данные выводятся на консоль
client.on('data',function(data) {
   console.log(data);
});

// При закрытии сервера
client.on('close',function() {
   console.log('connection is closed');
});
```

В разделе «Защита передаваемых данных», с. 190, рассматривается HTTPS — SSL-версия HTTP, а также `Crypto` и TLS/SSL.

Сокет UDP

TCP требует выделенного соединения между двумя конечными точками. UDP — протокол, не требующий соединения; это означает, что соединение между двумя конечными точками не гарантировано. По этой причине протокол UDP по надежности и степени защиты от ошибок уступает TCP. С другой стороны, UDP обычно работает быстрее TCP, что делает его более популярным для задач реального времени и таких технологий, как VoIP (Voice over Internet Protocol), в которых требования к соединению TCP могут отрицательно повлиять на качество сигнала.

Базовая функциональность Node поддерживает оба типа сокетов. Функциональность TCP уже была продемонстрирована ранее, теперь очередь UDP.

Для модуля UDP используется идентификатор `dgram`:

```
require ('dgram');
```

Чтобы создать сокет UDP, вызовите метод `createSocket` и передайте ему тип сокета — `udp4` или `udp6`. Также можно передать методу функцию обратного вызова для прослушивания событий. В отличие от сообщений, отправляемых по протоколу TCP, сообщения UDP должны отправляться в виде буферов, но не в виде строк.

В листинге 7.5 приведен демонстрационный код клиента UDP. В нем клиент обращается к данным через `process.stdin` и отправляет их через сокет UDP. Указывать кодировку строки при этом необязательно, потому что сокет UDP принимает только буфер, а данные `process.stdin` представляют собой буфер. Однако нам приходится преобразовывать буфер в строку методом `toString()` буфера, чтобы получить осмысленную строку для эхо-вывода данных вызовом `console.log()`.

Листинг 7.5. Клиент UDP для отправки сообщений, вводимых с терминала

```
var dgram = require('dgram');

var client = dgram.createSocket("udp4");

process.stdin.on('data', function (data) {
   console.log(data.toString('utf8'));
   client.send(data, 0, data.length, 8124, "examples.burningbird.net",
      function (err, bytes) {
        if (err)
          console.error('error: ' + err);
        else
          console.log('successful');
   });
});
```

Сервер UDP, приведенный в листинге 7.6, еще проще клиента. По сути, он всего лишь создает сокет, связывает его с конкретным портом (8124) и прослушивает событие `message`. При поступлении сообщения приложение выводит его, а также IP-адрес и порт отправителя с использованием `console.log`. Для вывода сообщения кодировка не обязательна — оно автоматически преобразуется из буфера в строку.

Связывать сокет с портом не обязательно. Тем не менее без связывания сокет будет пытаться вести прослушивание на каждом порте.